Solaiman E, Sfyrakis I, Molina-Jimenez C.

A State Aware Model and Architecture for the Monitoring and Enforcement of Electronic Contracts.

***In: 18th IEEE Conference on Business Informatics (CBI)*. 2016, Paris, France: IEEE**

DOI link to article:

http://doi.org/10.1109/CBI.2016.15

Date deposited:

05/09/2016

A State Aware Model and Architecture for the Monitoring and Enforcement of Electronic Contracts

Ellis Solaiman, *member, IEEE*
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
ellis.solaiman@ncl.ac.uk

Ioannis Sfyrakis
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
i.sfyrakis@ncl.ac.uk

Carlos Molina-Jimenez
Computer Laboratory
University of Cambridge
Cambridge, UK
carlos.molina@cl.cam.ac.uk

***Abstract*—Internet, Cloud, and IoT (Internet of Things) based business relationships involve electronic interactions that are normally regulated using Service Level Agreements (SLAs), and contracts that specify the rights, obligations, and prohibitions of the entities involved in the interactions. After a contract has been negotiated and agreed, all parties will need assurances that the service interactions comply with the clauses of the agreements between the parties, and that any violations are detected, prevented, and their causes identified. Because of the dynamic nature of emerging IoT and Cloud based relationships, there is a need for automated support for the monitoring and enforcement of service agreement policies. This paper develops a novel model for representing contract clauses using business rules that is specifically designed for contract compliance checking and enforcement. We identify what events need to be generated and captured from the underlying messaging middleware, and describe key design issues for a state aware contract monitoring and enforcement service.**

***Index Terms*—electronic contract, service level agreement, monitoring, enforcement, policies, web service, access control, research data**

I. Introduction

Advances in Internet and Cloud computing technologies have made it possible for businesses to provide increasingly sophisticated infrastructure and software services to their business partners and to their customers at affordable costs. Before a relationship between a service provider and a service consumer can commence, concerns such as security, and quality of the services provided, need to be agreed. Such details are normally formalized in the form of Service Level Agreements *(SLAs)* [1]. Service agreements explicitly define the permissible actions of the interacting parties, thus providing a legal basis for the resolution of any disputes. A Legal agreement can also be used as a guide for developing an electronic contract [2]. The main purpose of an electronic contract is to provide a mechanism that can automatically regulate (monitor and/or enforce) electronic service exchanges between contracted parties. Therefore ensuring that participants adhere to agreements in place, and that performed actions comply with various message timing and sequencing constraints.

The need for monitoring and enforcement mechanisms such as electronic contracts, has become more important with advances in paradigms such as *big data*, *cloud computing*, and the *Internet of Things (IoT)* [3]. Such advances have resulted in a phenomenal increase in the complexity of interactions between collaborators, business partners, and the distributed systems involved. Checking that such complex interactions correctly comply with any agreements in place manually, is almost an impossible task [4].

Developing mechanisms that are capable of monitoring and enforcing electronic contracts correctly is not easy. Previous work on contract monitoring has covered problems such as electronic contract representation and monitoring [5][6][7][8], contract verification [9][10], and automated testing [11]. In our previous work a contract monitoring service is mainly a passive observer that does not interfere with interactions between contracted parties. In this paper we focus our attention on the problem of contract enforcement (a key requirement identified in [8]). A contract enforcement service must be capable of ensuring that a business operation is executed only if it is contract compliant in accordance with the contract clauses.

The *contract compliance checker (CCC)* (Fig. 1), described previously in [8], is an independent contract monitoring service, which is event driven and state aware. When provided with an executable specification of a contract, it can be deployed by the contracted parties or by a third party. The CCC is able to observe and log relevant interaction events, which it processes to determine whether the actions of the business partners are consistent with respect to the rights, obligations, and prohibitions declared in the original legal contract. Namely, the CCC declares interaction events as either contract compliant (*CC*) or non contract compliant (*NCC*). As shown in Fig 1, business partners use a communication channel for exchanging their business messages. In addition they use a monitoring channel for notifying events of interest to the CCC. Notably, the figure shows that the CCC can cope with exceptions and failures, observing events that have been declared by the interacting parties as either *S (successful), TF (technical failure), or BF (business failure)*.

As a contract monitoring service, the CCC acts as a passive observer, which makes no attempt to influence the sequence and timing of message exchanges between business partners, or between the components of a distributed system. In this paper we explore key design requirements and present a

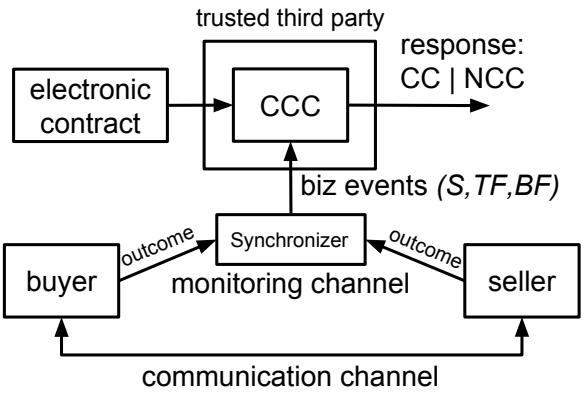

Fig. 1. The Contract Compliance Checker (CCC) monitor service.

model and enhanced implementation of the CCC that is able to act as a contract enforcement service, which is state aware and that directly influences the initiation of message exchanges and their outcomes.

Therefore this paper makes the following key contributions; we describe a model for contract enforcement while explicitly taking into consideration the rights, obligations, and prohibitions within the policies of the electronic contract as the interacting parties move from state to state; and to enable this functionality we present an architecture for a proof of concept contract enforcer service that is capable of intercepting the initiation events that precede the execution of operations, and prevent execution of operations that are none contract compliant. The enforcer service is also proactive whereby it can remind partners of their obligations well before deadlines expire.

The remainder of the paper is organized as follows: In Section II we describe key electronic contracting concepts, and our *ROP* ontology with the aid of a simple example. In Section III, we analyze the requirements for a contract enforcement service with the aid of a research data management scenario. Implementation details of the contract monitoring and enforcement service are presented in Section IV. We place our work within the context of current research in Section V. We draw conclusions and motivate further research in this direction in Section VI.

II. Background

In order to elaborate key electronic contracting concepts, we present a simple scenario. Let us assume that Fig. 1 describes a relationship where two organisations, a Buyer and a Seller (a store), agree to a business contract. Below are some of its clauses:

1) *The buyer can place a **buy request** with the store to buy an item.*
2) *The store is obliged to respond with either **buy confirmation** or **buy rejection** within 3 days of receiving the buy request.*
    a) *No response from the store within 3 days will be treated as a buy rejection.*
3) *The buyer can either **pay** or **cancel** the buy request within 7 days of receiving a confirmation.*
    a) *No response from the buyer within 7 days will be treated as a cancellation.*

The clauses of such a legal agreement should take into consideration all relevant business operations (shown in bold in the contract text). A business contract specifies a well defined list of business operations. A business operation is a business activity which the participants are able to perform under certain conditions. In the CCC, business operations are used to formally define the vocabulary (alphabet) of the interaction. We use $B = \{bo_1, ..., bo_n\}$ to represent all the valid business operations in the contract. The clauses of our example contain five business operations {*buy request, buy reject, buy confirmation, buy payment, buy cancellation*} shown in bold in the English text of the contract. The buyer and seller are regarded as role players interested in executing the operations is a shared fashion. The set of valid role players is represented by $RP = \{rp_1, ...,rp_n\}$.

The execution of each business process generates an individual outcome event which is passed to the synchronizer shown in Fig. 1 through the monitor channel. The synchronizer integrates the pair of individual outcomes from each side into a single business event. This business event is sent to the CCC. As a monitor, the responsibility of the CCC is to determine whether a given event presented to it represents the notification of a contract compliant operation *CC*, or a none contract compliant operation *NCC*. To be able to make this determination, the CCC keeps track of the state of interaction as a *Finite State Machine (FSM)* with states being determined by enabling and disabling the current rights, obligations and prohibitions of the role players in force.

A. ROP Ontology

A contract distinguishes operations as *Rights*, *Obligations*, and *Prohibitions* (the *ROP* set). A *Right* is an operation that a party is allowed to perform under certain conditions, an *Obligation* is an operation that a party is expected to do under certain conditions, and a *Prohibition* is an operation that a party is not allowed to do under certain conditions.

We define an individual right r_i, obligation o_i or prohibition p_i as a set of operations where: $r_i \subseteq B$, $o_i \subseteq B$, and $p_i \subseteq B$. For a particular role player *RP*; $R_{rp} = \{r_1,...,r_n\}$; $O_{rp} = \{o_1,...,o_n\}$; and $P_{rp} = \{p_1,...,p_n\}$, represent the sets of rights, obligations, and prohibitions currently assigned to the role player *RP* respectively. The sets of rights, obligations, and prohibitions of an *RP* are represented as ROP_{rp}.

B. Electronic Contracts

The electronic contract designer is able to use the legal contract in order to accurately identify and extract the *ROP* set attributed to the business partners, and to specify the rules which operate on the *ROP* set. Rule implementation requires an appropriate specification language; contract rules written

for the CCC service are currently realized using the *Drools Rule Language* [12], wrapped within our *ROP* ontology.

An example of a rule that deals with receipt of a *buy request* event by the CCC, written using Drools can be seen below. Line 5 checks that the *buyRequest* operation is a right that the buyer is currently allowed to perform. If so then *buyRequest* is declared by the CCC as contract compliant (line 13). This operation is also removed from the buyer's ROP set (line 8), meaning that the buyer no longer has a right to perform this operation. At lines 10 and 11, the seller is given an obligation to perform one of 2 operations: *buyConfirm*, or *buyReject*.

```
1 rule "Buy Request Received"
2 //Verify type of event, originator, and
    responder
3 when
4  $e: Event(type=="BUYREQ", originator=="
    buyer", responder=="store", status=="
    success")
5    eval(ropBuyer.matchesRights(buyRequest))
6 then
7  //Remove buyer's right to place other Buy
    Requests
8  ropBuyer.removeRight(buyRequest, seller);
9  //Add seller's obligation to either accept
     or reject order
10 BusinessOperation[] bos = {buyConfirm,
    buyReject};
11 ropSeller.addObligation("React To Buy
    Request", bos, buyer, 60,2);
12 System.out.println("* Buy Request Received
     rule triggered");
13 responder.setContractCompliant(true)
14 end
```

Each of the business operations in bold within the contract clauses of our example, has a rule such as the one shown above. Typically, for an activity, each business partner can have several rights, obligations, and prohibitions in force. Once an electronic contract specification has been completed, it can be loaded into the CCC for deployment. As operations are executed, and events are received by the CCC; rights, obligations, and prohibitions are granted and revoked as specified by the rules. Therefore when the CCC acts as a contract monitor, a right obligation or prohibition, is in one of two states only: *inactive* or *active*.

C. Contract Compliance

The CCC processes each event to determine if it is contract compliant *(CC)* or none contract compliant *(NCC)*. The execution of a business operation is said to be *CC* if it satisfies the following three conditions and is said to be *NCC* if it does not:

1) $bo_i \in BO$; the business operation matches an operation within the set of business operations expected by the CCC.

2) $bo_i \vdash ROP_{rp}$; the business operation matches the *ROP* set of its role player (meaning, the role player that performed the operation has a right/obligation/prohibition to perform that operation). By "match", we mean that for a valid business operation bo_i, and a particular role player's *ROP* set; ROP_{rp} where: R_{rp} = $\{r_1,...,r_m\}$, O_{rp} = $\{o_1,...,o_m\}$, P_{rp} = $\{p_1,...,p_m\}$, and $m \geq 1$, their relationship should be that: $bo_i \in r_j$ or $bo_i \in o_j$ or $bo_i \in p_j$, where $1 \leq j \leq m$.

3) the business operation must also satisfy the constraints stipulated in the contractual clauses. An example of a constraint is the seven day deadline in clause 3 of our contract example shown earlier.

We also consider that the execution of a given sequence of operations is *NCC* if it includes one or more operations that are flagged by the CCC as *NCC*. A sequence of operations is also known as an *execution sequence* or *execution trace* and drives the choreography from its initial state to a final state.

D. Exception Handling

To ease the introduction of basic concepts, our legal contract example deals with successful outcome events only. However, the CCC contract monitoring service is also able to observe outcome events that include exceptional circumstances [13]. Following the ebXML standard [14], at the end of a business conversation, each party independently declares an execution outcome event from the set {*Success(S)*, *BizFail(BF)*, *TecFail(TF)*} as shown in Fig. 1. *Success* events model successful execution outcomes. *TecFail* models protocol related failures detected at the middleware level, such as a late, or a syntactically incorrect message. *BizFail* models semantic errors in a message detected at the business level, e.g., the credit card details extracted from the received payment document are incorrect.

E. Discussion

It is clear that the information available to the CCC can also be used by the interacting partners to maintain consistency of the overall state of interaction between all parties that are involved. For example, a business partner can use the termination status *(CC — NCC)* received from the CCC, in order to drive its execution of business processes, thereby eliminating (or substantially reducing the occurrences of) situations that arise due to divergence in views. A natural extension therefore would be to convert the CCC to a contract enforcer that prevents the partners from performing prohibited operations. The *timer* component within the CCC architecture can also be used to identify obligations that have not been fulfilled by certain deadlines, and to proactively inform affected participants.

III. Contract Enforcement

The CCC presented in the previous section acts as an observer that does not interfere with the actions of the interacting parties. For the CCC to act as an enforcer it must be intimately involved in the execution of operations between all parties. In the following section we discuss the requirements for a contract enforcement service using a *Research Data (RD)* storage and access control scenario.

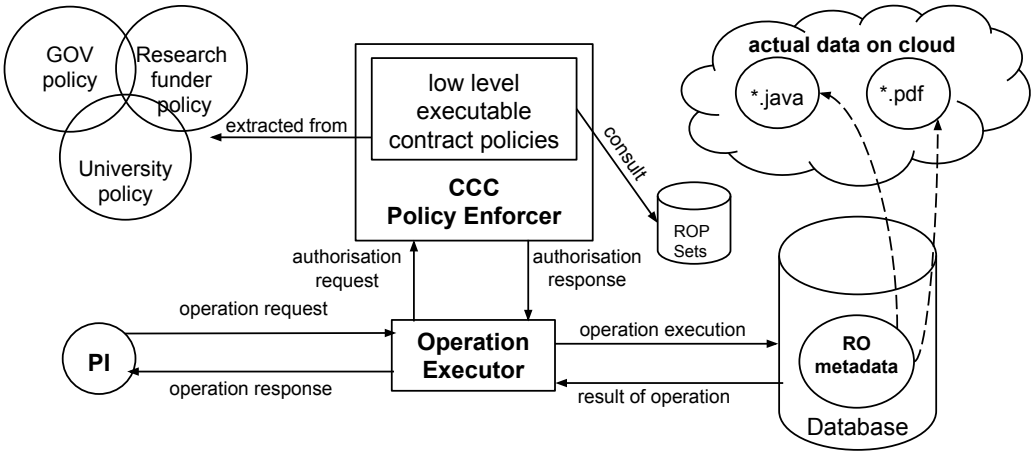

Fig. 2. High level view of CCC Policy Enforcer and Operation Executor.

A. Policy Enforcement Scenario

Scientific activities normally result in the production of research data that can be used to support scientific claims. By *Research Data (RD)*, we mean data produced from research activities such as computer programs, genome sequences, data recorded during archaeological excavation, data collected about atmospheric conditions, and so forth. As well as traditional publications such as journal, conference papers and technical reports. It has become common practice to store research data in electronic mediums, such as plain files or database records. The data can be stored locally within an organization, on a public cloud, or using a hybrid cloud solution depending on the nature of the data being stored and various security and privacy needs. As a general rule, research data is owned by the institution (for example a university) that employs the scientists involved in the research. Thus research institutions are legally responsible for guaranteeing that RD generated under their auspices comply with a number of policies. Intuitively speaking, RD policies are meant to regulate the management of RD to guarantee that it is kept for as long as necessary, and made widely accessible to view and reuse under the observance of some requirements. Typical requirements are protection of intellectual property and privacy, and accuracy of the RD so that it allows reproducibility (or at least repeatability) of the original scientific experiment [15].

The following is a typical policy imposed on RD at a research university: *"The University has the right of unfettered access to RD generated from research activities conducted by its research staff"*. As a second example, take the following policy imposed on RD at the University of Pittsburgh: *"RD is retained for a minimum of seven years after the final reporting or publication of a project"*.

Legally speaking, universities are responsible for ensuring that RD policies are observed. They are expected to devise mechanisms that guarantee that the operations executed by RD users against RD comply with the RD policies in force.

The enforcement of RD policies is a complex and currently a poorly understood topic that deserves research attention. For instance it is not surprising that in order to tackle this issue in the United Kingdom, the *Digital Curation Centre (DCC)* was created in the UK in 2010 [16] to provide expert advice and practical help in RD management to UK higher education institutions. For example, RD needs to comply with policies stipulated independently by several bodies such as the research institution that owns the RD, the sponsors of the project that generated the RD, as well as governmental policies.

In this paper we argue that this problem can be addressed by the use of an automatic policy enforcer service that regulates the operations executed against abstract structures of research data, known as *Research Objects (RO)* [17]. A simplified example of research data policy is given next to support our discussion:

1) *PI (Principal Investigator) has the right to upload RD to the RD repository after his registration to the repository.*
2) *If the size of RD PI uploads is less than 10 MB.*
   a) *PI is prohibited from deleting the RD.*
   b) *PI has a right to update the latest publication date of the RD he/she has uploaded within 2 years.*
   c) *PI has the right to download the RD he/she has uploaded.*
3) *If the size of RD uploaded by PI is larger than 10 MB.*
   a) *PI is obliged to specify the raw data and tools for reproducing this RD.*
   b) *PI has an obligation to delete the RD from the repository within one month after he fulfills his obligation of (a).*

Fig 2 shows three possible policy stakeholders *(University, Research funder such as the EU, and a national government)*. But in practice, more stakeholders such as industrial partners can be involved. The figure also shows a high level view of

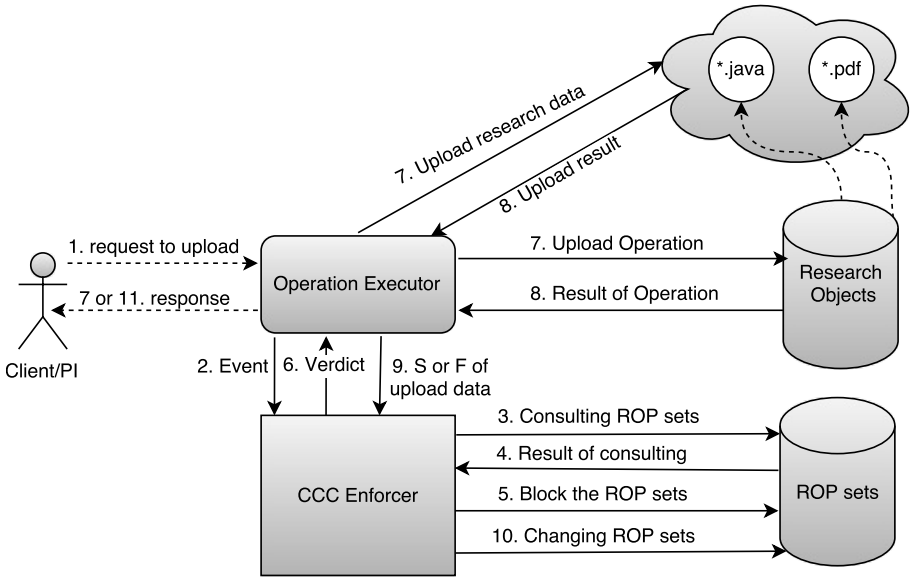

Fig. 3. Typical interaction with the CCC Enforcer.

the CCC when acting as an enforcer with two main components, the *Policy Enforcer*, and the *Operation Executor*.

The state of the a RO can be changed by operations requested by RD users. In Fig 2, the PI (Principal Investigator) is shown placing an operation request. An operation is executed by the *operation executor* only if the operation is authorized by the *policy enforcer*.

As with the contract example shown in Section II, a research data policy contract essentially specifies the rights, obligations and prohibitions of the users involved. In the figure, research objects *(RO)* are stored in a meta-data database. A research object contains links to the actual research data represented by it. An operation on a *RO* therefore could be mapped to the actual research data. The three high level policies of our example are integrated into low level policies (such as the *Drools* rules described in Section II-B), which are executable by the *policy enforcer*, and must be conflict free. The *policy enforcer* authorizes a request only if it finds it to be compliant with the list of low level policies derived from the overall RD policy description.

By enforcement of RD policies, we mean that the policy enforcer should be able to prevent those operations which violate the RD policies. So the enforcer has to decide whether an operation on research data should be permitted before it is performed. If the operation violates any policies, then permission should be denied.

B. Example Execution

An example execution of the previous scenario is shown in Fig. 3. The CCC web service provides research data storage service on the cloud and the RD stored in it are enforced by RD policies shown earlier. Here is a description of the steps shown in the figure:

1) *In Fig. 3, the client (for example the PI) sends a request to upload a research object to the database.*
2) *The* operation executor *constructs an event object for the request received. This event includes information about the client and the performed operation. This object is then passed to the policy enforcer.*
3) *After receiving the event object, the policy enforcer logs the event and checks for policy compliance of the* upload *operation according to the coded RD contract policies. The policy enforcer also checks the ROP sets.*
4) *The policy enforcer then decides whether this upload event is contract-compliant (CC) or not (NCC).*
5) *Importantly, the policy enforcer then blocks the ROP sets, and further operations are temporarily prevented in order to avoid causing inconsistencies in the state of the ROP set while the current operation is in progress.*
6) *The policy returns its verdict on the event to the operation executor.*
7) a) *If the verdict is that the operation is Contract Compliant, the executor will attempt to upload the research object in the database.*
   b) *If the verdict is None Contract Compliant, the executor will respond to the user informing them that the operation is not allowed.*
8) *The executor gets the execution result of the upload operation.*
9) a) *If the research object is uploaded successfully, the executor will pass a succeed event (S event) to the policy enforcer to notify it that the upload operation was executed successfully.*
   b) *If the upload operation fails, then the executor will send a failed event to the policy enforcer to inform it that the delete operation has failed.*

10) *The policy enforcer will change the ROP set of the user according to the result of the upload action and according to the requirements of the RD policies.*
11) *Finally, the executor returns the execution result to the user.*

C. Enhancing the ROP ontology for contract enforcement

Within the CCC monitoring service, each of a role player's rights obligations or prohibitions, need only be in one of two possible states: either *active* or *inactive*. For the CCC to operate as an enforcer these two states are inadequate. Because the policy enforcer determines the contractual compliance of an operation before its execution (it also has to detect potential execution failures), we introduce a new state for describing a right, obligation or prohibition currently in execution called *inExecution*. When a right, obligation or prohibition is matched by an operation, its state will become *inExecution* until the execution result of the operation is determined. This is to prevent the modification of the ROP set by other operations (parallel execution attempt) while the current operation outcome is being determined by the CCC. This state is then changed by the policy enforcer according to both the execution result, and according to the contract policy. For example, the RD contract policies of our previous example specify that a PI has an obligation to *specify the raw data and tools for reproducing this RD if the size of the RD uploaded is larger than 10 MB*. According to the policy, if the PI fulfills this obligation, the obligation should be removed from the ROP sets of the PI which means that the PI no longer has this obligation. However, if the PI fails to fulfill this obligation, then the obligation should remain imposed because the data and tools have not been uploaded.

Fig. 4 illustrates how the state of an obligation to delete a file changes through time where; *t1* is the time at which the policy enforcer decides if the delete operation is policy-compliant or not, and *t2* is the time at which the policy enforcer receives the result of executing the delete operation.

First, the obligation to delete the file is imposed on the PI. Then the state of the obligation to delete the file changes to *inExecution* when the policy enforcer considers the delete operation policy-compliant. During the state of *inExecution*, this ROP set is temporarily unavailable. If the PI tries to send another request to delete the same file when the obligation to delete the file is *inExecution*, this attempt will be blocked.

After a successful execution of the *delete* operation, the state of the obligation is set to *fulfilled*, which means that the obligation is honored as shown Fig. 4 a). If the result of execution fails, then the state of the obligation will be changed back to *imposed* so that PI can try to delete the file again as shown in Fig. 4 b).

The significance of introducing *inExecution* is that sometimes performing an operation takes a relatively long time, and during this time the PI should not be able to re-perform the same operation until it has completed. Also, when the policy enforcer receives an operation execution result, it has

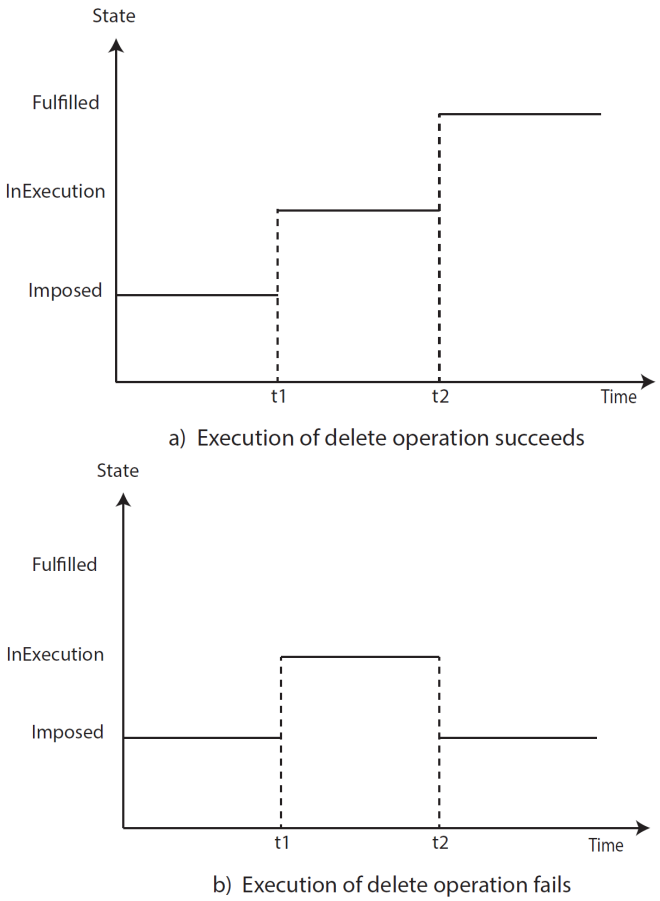

Fig. 4. The state of the delete obligation changes through time.

to distinguish which rights, obligations or prohibitions have definitively been previously executed.

In Fig. 4 the state *imposed* within the enforcer service is the same as the *active* state of the CCC monitor service. In addition, we introduce the state *fulfilled* if an obligation or prohibition has been honored. A right has no state of *fulfilled* as a right can potentially be performed many times. But a particular obligation or prohibition can only be performed once. Therefore, we use a counter to keep track of how many times a right has been performed. Fig. 5 shows the states that rights, obligations, and prohibitions can go through during their life-time. A new state called *violated* is also introduced which means that a right, obligation or prohibition has not been fulfilled after a particular deadline. Because the deadline for a right or prohibition may not necessarily be triggered, the state *violated* is optional for rights and prohibitions. The state of an individual right r, obligation o and prohibition p is represented by S_r, S_o, and S_p respectively.

D. Contract Compliance within the Enforcer

For an operation op_i by a role player rp, to be contract-compliant, the event for representing its execution, $e\{rp, op_i, status, time\}$ must satisfy the following conditions:

	Right	Obligation	Prohibition
State	Imposed Inactive InExecution Violated (optional)	Imposed Inactive InExecution Violated Fulfilled	Imposed Inactive InExecution Violated (optional) Fulfilled

Fig. 5. Possible states for Rights, Obligations, and Prohibitions.

1) $rp \in$ *RP*; the role player rp who initiates an operation op_i must belong to the set of valid users RP.
2) $op_i \in$ *OP*; an operation op_i must be a known operation. In other words, it must be one of operations specified in the set of operations OP.
3) $op_i \vdash ROP_{rp}$; an operation op_i must match an operation within the initiating role player's ROP set. Meaning that the user who initiates the operation op_i must have a right, obligation or prohibition to perform it.
4) $S_e \in \{notExecuted,\ success,\ failed\}$ the status of event must be $notExecuted$, $success$ or $failed$. Any status except the three will be treated as none contract compliant (*NCC*).
5) time constraint (optional). The time of occurrence of an event must be before the deadline of the matched right, obligation or prohibition.
6) history of event (optional). Whether an event is policy-compliant or not may depend on the occurrence of other events.
7) any other constrains specified in the contract (optional).

To illustrate the last three constrains, here is an example of a RD (research data) policy with two clauses:

A research paper can be deleted from the system if:

1) *Its size is larger than 1024 MB (other constraint).*
2) *After 1 year of uploading (time constraint).*
3) *The raw data and software for reproducing it are also stored in the system (history of event constraint).*

IV. Architecture of the Policy Enforcer

A high level view of the CCC is shown in Fig. 6. It consists of two layers: The *CCC Engine* (The *Logical Layer*), and the *CCC Service* (The *Presentation Layer*). The *CCC Engine* is responsible for processing business events and for determining whether they are contract compliant or not. The *CCC Service* is an interface to the *CCC Engine*, it is used for delivering business events to the CCC, and for collecting the corresponding responses. In addition, the *CCC Service* can be used by the rule administrator for loading and editing the rules that represent the contract. The functionality of the architecture is as follows: An event is received through the *monitoring channel* as an XML document that includes the names of the initiator, the operation, and its outcome from the set: *(Success, BizFail, TecFail)*:

```
<event>
  <originator>PI</originator>
  <type>Upload</type>
  <status>success</status>
</event>
```

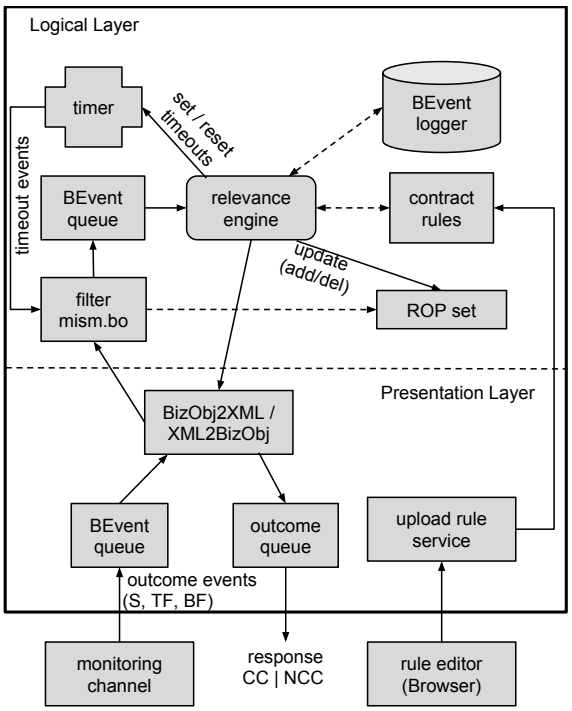

Fig. 6. Architecture of the Contract Compliance Checker (CCC).

The XML document representing the event is passed to the *BEvent queue*. Business events are retrieved, and converted using the *xml2BizObj/BizObj2xml Converter* from their XML format into business event objects. Events are then passed to the *CCC Engine*. The *filter mism.bo*, discards mismatched business events that are not among the permitted events defined within the ROP set. Business events that pass this filter are inserted into the *BEvent queue*. All deadlines are set and reset by the *relevance engine*, and enforced by the *timer*. Timeout events are added to the *filter mism.bo* as required by the contract, and are examined by the filter to decide if *bevents* are mismatched. For example, receiving an *upload* event from the *PI* after a 3 day deadline has elapsed, will be treated as mismatched. The *relevance engine* removes a business event from the head of the *bevent queue* and compares it to the rules stored in the *contract rules*. Rules that match the *bevent* under examination are triggered to determine if their conditions are satisfied. The actions of the rules whose conditions are satisfied are executed, and this may alter (*add/del*) the current state of the *ROP sets*. The *bevent* is then stored in the *BEvent logger* as a record for any future dispute resolution. The *relevance engine* eventually declares the business event either *CC* or *NCC* and produces a response as a business object, which is sent out to the *Presentation Layer*. The business object passes through the *xml2BizObj/BizObj2xml Converter*, where it is serialized into

an XML message of the following format:

```
<result>
 <contractcompliant>true|false
 </contractcompliant>
</result>
```

If the decision of the CCC Enforcer is that the event is none contract compliant then an extra element is attached to the response event. This element outlines the reason the event is not contract compliant. An xml representation of such a response event is displayed below:

```
<result>
 <contractCompliant>false
 </contractCompliant>
 <message>Obligation to upload before
    deadline is violated
 </message>
</result>
```

The *xml2BizObj/BizObj2xml Converter* inserts the response into the *outcome queue*, which can be accessed by the contracted parties. The *Presentation Layer* allows a "rule manager" to update the *contract rules* at run time. For this purpose, rules can be edited using the *rule editor (in a browser)* and sent to the *rule upload service* as a conventional RESTful POST operation. The *rule upload service* is responsible for producing a *drl* (Drools) file (for example *new–rules.drl*) from the payload of the POST operation, and for uploading it to the *CCC Logical Layer* to replace the *Contract Rules*.

The *CCC Logical Layer* is implemented using JBoss's Drools rules engine [12]. The Drools rules engine powers the decision making capabilities of the *relevance engine*. The *relevance engine*, acts as a wrapper for the Drools rule engine and its responsibilities include the initialisation of the contract, as well as the addition and processing of events received from the *Presentation Layer*.

The *Presentation layer*, exposes the CCC as a RESTful web service. Its aim is to enable the exchange of XML event messages between the CCC and the contracted clients, and to ease the editing and update of the contract rules. The *Presentation Layer* is implemented using the JBoss Enterprise Application Platform (EAP), [18]. The *BEvent queue* and the *outcome queue*, are implemented using JBoss's HornetQ (a message oriented middleware layer), and using the Java Message Service (JMS) API. A Message Driven Bean (MDB) receives business events from HornetQ and passes them to the *XML2BizObj/BizObj2XML* converter, which is implemented using Java. The *upload rule service* is part of the Drools Workbench– a web authoring and rules management application.

V. Related Work

Research work on the monitoring and enforcement of cross-organizational interactions between parties was pioneered by Minsky [19] with work on Law Governed Interaction (LGI). LGI is a *law enforcer* that regulates the interaction between autonomous and distributed agents linked by a communication network. A controller instrumented with the law (e.g., contractual clauses) is placed between each agent and the network to intercept and rule on incoming or outgoing messages that are incompatible with the law, keep the agent state in synchrony with other agents, verify certain conditions, and execute relevant actions to enforce the law imposed on the agent. The LGI system architecture is peer-to-peer in the sense that each participant is required to run an instance of LGI, whereas we have examined compliance checking from the view point of a *trusted third party*. Further, unlike our work, timing and message validity constraints that are an essential part of on-line messaging are not considered in LGI.

The notion of rights, obligations and prohibitions was introduced in [20]. A useful summary about various issues involved in contract management is provided in [21].

We are not the first to suggest an event centric approach to model contracts. In [22] the authors describe an obligation enforcer focused on the enforcement of resource usage policies such as *No execution should last more than one second*. It is designed under the assumption that all actions are allowed and (if necessary) compensatory; likewise all resources are assumed to be preemptive (e.g., *Abort a given job after 3 sec of execution to free CPU*). Consequently, unlike in our work, the notion of rights and prohibitions are not of concern. The architecture of the obligation enforcer of [22] bears some similarities to our CCC, except that pending obligations are recorded in a history log whereas in our work we express them explicitly in ROP sets, and therefore our CCC is able to automatically detect the none fulfillment of obligations by deadlines specified within the Drools rules. In [23], an event based SLA compliance monitoring framework is presented where the monitoring platform is capable of reacting to compliance violations as soon as they happen – as we do using our CCC. The paper also describes the advantages of immediate reactions to violations such as an enterprise avoiding or at least minimizing penalty fees for SLAs, in addition to potential reduction of operational costs. The paper goes on to highlight important future work on including forecasting capabilities into their compliance model, which we address within our model using the *timer* component, which is capable of generating events reminding contracted parties of their rights, prohibitions, or obligations well before their deadlines.

A general overview of the four phases of electronic contracting (information phase, pre-contracting, contracting, and enactment) that an electronic contracting process involves is presented in [24]. With respect to this taxonomy, our work focuses on the enactment phase and in particular, on the monitoring, enforcement, and control activity. Our aim is to provide a concrete solution to monitor and when necessary enforce interactions in various distributed application settings, and also collect historical records that can assist

in off-line evaluation of contract compliance and in dispute resolution.

VI. Conclusions and Future Work

The *CCC (Contract Compliance Checker)* is a state aware event driven mechanism that when supplied with an appropriate electronic contract specification, is capable of determining the contractual compliance of electronic exchanges between collaborators. In this paper we have presented an enhancement to the CCC service, and to our ROP ontology, enabling it be deployed as an enforcer service. We have determined the requirements for such a service using a simplified but realistic Research Data access control scenario.

Unlike the CCC monitoring service which acts only as an observer, a contract enforcement service needs to be intimately involved in the execution of operations, actively preventing the initiation of some operations, and proactively detecting and informing affected parties of the none-fulfillment of obligations. The CCC Enforcer observes and logs relevant interaction events, which it processes to determine whether actions are contract compliant (*CC*) or none contract compliant (*NCC*).

An important item for future work is the development of tools for the automatic verification and testing of coded electronic contracts. As the complexity of interactions in domains such as IoT, and cloud computing increases, naturally so do the electronic contracts. Such complexity makes it much more likely for conflicts between the clauses of contracts to occur (for example mutually prohibiting and obliging an operation). Thus the need for verification, and for the development of tools (such as [25]) that makes verification easier for none computing experts is extremely important.

References

[1] D. Kyriazis, "Cloud computing service level agreements - exploitation of research results," *European Commission*, 2013.

[2] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne, "Contract representation for run-time monitoring and enforcement," in *2003 IEEE International Conference on E-Commerce (CEC 2003)*. IEEE, 2003.

[3] E. Solaiman, R. Ranjan, P. Jayaraman, and K. Mitra, "Failure monitoring in the internet of things application ecosystems: Cloud to edge," *IT Professional IEEE Computer Society*, 2016, in press.

[4] C. Molina-Jimenez, S. Shrivastava, and S. Wheater, "An architecture for negotiation and enforcement of resource usage policies," in *IEEE International Conference on Service Oriented Computing & Applications (SOCA)*. IEEE, 2011.

[5] C. Molina-Jimenez, S. Shrivastava, E. Solaiman, and J. Warne, "Run-time monitoring and enforcement of electronic contracts," *Electronic Commerce Research and Applications*, 2004.

[6] M. Strano, C. Molina-Jimenez, and S. Shrivastava, "A rule-based notation to specify executable electronic contracts," in *Rule Representation, Interchange and Reasoning on the Web: International Symposium (RuleML)*. Springer-Verlag, 2008.

[7] G. Governatori, Z. Milosevic, and S. Sadiq, "Compliance checking between business processes and business contracts," in *10th Int'l Enterprise Distrib. Object Computing Conf. (EDOC'06)*. IEEE CS, 2006, pp. 221–232.

[8] C. Molina-Jimenez, S. Shrivastava, and M. Strano, "A model for checking contractual compliance of business interactions," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 276–289, 2012.

[9] E. Solaiman, C. Molina-Jimenez, and S. Shrivastava, "Model checking correctness properties of electronic contracts," in *International Conference on Service Oriented Computing (ICSOC03)*. Springer, 2003.

[10] A. Abdelsadiq, C. Molina-Jimenez, and S. Shrivastava, "A high level model checking tool for verifying service agreements," in *The 6th IEEE International Symposium on Service-Oriented System Engineering (SOSE 2011)*. IEEE, 2011.

[11] E. Solaiman, I. Sfyrakis, and C. Molina-Jimenez, "High level model checker based testing of electronic contracts," *Cloud Computing and Services Science, Springer-Verlag*, 2016.

[12] RedHat, *"Drools"*, http://www.drools.org/, 2016.

[13] C. Molina-Jimenez, S. Shrivastava, and M. Strano, "Exception handling in electronic contracting," in *IEEE Conference on Commerce and Enterprise Computing (CEC). 2009, Vienna, Austria*. IEEE, 2009.

[14] OASIS, *ebXML Business Process Specification Schema Technical Specification v2.0.4*, Available: http://docs.oasis-open.org/ebxml-bp/2.0.4/OS/spec/ebxmlbp-v2.0.4-Spec-os-en.pdf, 2006.

[15] P. Missier, S. Woodman, H. Hiden, and P. Watson, "Provenance and data differencing for workow reproducibility analysis," *Concurrency and Computation: Practice and Experience*, 2013.

[16] DCC, "Digital curation center's home page," http://www.dcc.ac.uk, Last accessed 2016.

[17] S. Bechhofer, I. Buchan, D. D. Roure, P. Missier, J. Ainsworth, J. Bhagat, P. Couch, D. Cruickshank, M. Delderfiel, I. Dunlop, M. Gamble, D. Michaelides, S. Owen, D. Newman, S. Sufi, and C. Goble, "Why linked data is not enough for scientists," *Future Generation Computer Systems*, 2013.

[18] RedHat, *JBoss Enterprise Application Platform*, http://www.redhat.com/en/technologies/jboss-middleware/application-platform, 2016.

[19] V. Ungureanu and N. H. Minsky, "Establishing business rules for inter enterprise electronic commerce," in *14th International Symposium on Distributed Computing (DISC00)*, 2000, pp. 179–193.

[20] H. Ludwig and M. Stolze, "Simple obligation and right model (sorm)-for the runtime management of electronic service contracts," in *2nd Intl Workshop on Web Services, eBusiness, and the Semantic Web (WES03) LNCS*, vol. 3095, 2003, pp. 62–76.

[21] T. Hvitved, "A survey of formal languages for contracts," in *n Fourth Workshop on Formal Languages and Analysis of Contract Oriented Software (FLACOS10)*, 2010.

[22] P. Gama, C. Ribeiro, and P. Ferreira, "Heimdhal: A history-based policy engine for grids," in *Sixth IEEE Intl Symp. Cluster Computing and the Grid (CCGRID 06)*. IEEE, 2006.

[23] R. Thullner and S. Rozsnyai, "Proactive business process compliance monitoring with event-based systems," in *Proc. 15th IEEE International Enterprise Distributed Object Computing Conference*, 2011.

[24] S. Angelov and P. Grefen, "Supporting the diversity of b2b econtracting processes," *International Journal of Electronic Commerce*, 2008.

[25] E. Solaiman, W. Sun, and C. Molina-Jimenez, "A tool for the automatic verification of bpmn choreographies," in *IEEE 12th International Conference on Services Computing (SCC)*. IEEE, 2015.